Diagnosing intrusions in Android operating system using system flow graph

Radoniaina Andriatsimandefitra, Valérie Viêt Triêm Tông, Ludovic Mé

CIDRE research group, Supélec, avenue de la Boulaie, 35576 Cesson-Sévigné

firstname.lastname@supelec.fr

Résumé – Le suivi de flux d'information est une technique qui consiste à surveiller les actions d'une (ou plusieurs) application(s) pour connaître comment ses (leurs) informations se disséminent dans un système d'exploitation. Nous avons montré dans des travaux précédents comment le suivi de flux d'information permet de détecter des applications malicieuses et nous nous intéressons ici à l'utilisation de ces techniques pour proposer un diagnostic des intrusions détectées. Nous définissons la notion de *graphe de flux système* qui exprime comment l'exécution d'une application mène à la dissémination d'information dans le système. Ce graphe de flux système décrit donc le comportement externe de l'application et sa construction ne nécessite aucune connaissance du fonctionnement interne de l'application. Ce graphe est obtenu à la fois par une configuration particulière du moniteur de flux d'information et par les journaux produits par ce moniteur. Nous présentons en fin d'article l'étude d'une application malicieuse polluant des plateformes de téléchargement d'application alternatives à *Google Play*.

Abstract – Monitoring information-flow consists in observing actions performed by an application to deduce how the information from the application disseminates within an operating system. We showed in previous works how this monitoring allows to detect malicious applications and we focus in this paper on its use to build a diagnostic of system-intrusions. We define a system flow-graph as a structure that describes the external behaviour of an application that leads to the information dissemination within the system. The system flow-graph thus describes the external behaviour of the application and its construction requires no knowledge about the inner-working of the application. The graph is built from a particular configuration of an information-flow monitor and the logs it produces. We present at the end of this paper an analysis of a malicious application discovered in third-party alternatives of *Google Play*.

A need for information flow control

Android is an operating system made for mobile devices (smartphones and tablets). Due to its widespread adoption and the sensitive nature of data it may contain, Android became the target of increasing malicious applications [1]. As pointed out in different studies [2, 3], Android security mechanism is not efficient to protect users and their sensitive pieces of information from malware. Security groups have thus worked on security extensions for Android to improve its security level. For example, Google security team has developed a tool, Google bouncer [4], that analyses applications published on Google play to detect which one are malicious. Google bouncer compares the application with known malicious applications and runs it on an Android emulator to detect malicious behaviour. We believe that it is a good step to security enhancement on Android devices. However people have proven that analysis in this emulated device can fail to detect misbehaviours [5, 6]. For instance, malicious applications can perform fingerprinting to detect if their running environment is a real device or not. When they detect an emulator, they do not misbehave. Monitoring of suspicious applications appears to us as an interesting idea but we claim that this monitoring should not be made on an emulator. We suggest to monitor the application inside their normal environment and we prefer to adopt information flow monitoring. Information flow monitoring uses tainting techniques to deduce how information are spread in the environment. The environnement is here the entire operating system. Information-flow monitoring has been used in a lot of work to protect [8, 9] a system from attacks or to detect them [2, 10, 11]. We believe that information-flow monitoring should not be limited to detection but should be extended to diagnose purposes too. In this paper, we propose a methodology to diagnose integrity violation on mobile devices by exploiting information-flows tehcniques. We build our work around a model presented in [13] and we use the corresponding tool named Blare that implements this model. We explain how we observe information flow from a third-application within the system and build the corresponding system flow-graph. The flow graph can be used to analyse the application behaviour and thus gives an early diagnosis.

In the following section we briefly present some inter-

esting previous works related to the use of information flows for attack detection, then section presents our methodology for detection and diagnostic of security violation.

Related works on information flow analysis

In order to know if applications from Google Play leak sensitive data out of the device, Enck et al. built TaintDroid [2], a modified version of Android that monitors flows of sensitive information. For that purpose, they selected pieces of data they judged sensitive on Android devices (e.g phone identifier) and labelled their source-container. When an application access to a labelled source TaintDroid considers that the application has retrieved a labelled content, and thus propagates the label. In TaintDroid, each program variables, files and IPC messages can be labelled. TaintDroid raises alerts when it detect that sensitive data have left the device. They selected thirty of the most popular applications from Google Play for their study. Their analysis showed that more than two of a third of them send sensitive data to remote entities (without any notification to the user).

Egele et al. also studied privacy leaks performed on iOS devices. They presented a tool named PiOS [10] that does static analysis on applications code to analyse how sensitive information flow in the application and detect an eventual privacy leak. Like Android, iOS proposes a framework that gives third-applications an access to sensitive data. Information sources are the functions that give access to sensitive-data and sinks are the functions used to communicate with remote entities. The privacy leaks that PiOS looks for is flow of sensitive information from these sources to sink parameters that serve to communicate with remote entities. The system was tested with existing applications from Apple Store and gave results similar as TaintDroid's. More than a half of studied applications leaked device identifier to remote entities.

Using similar approach, Yin et al. developed Panorama [11], a system that captures information flow at hardware level for malware detection and analysis. To detect malicious applications, authors run program samples in a controlled environment where tainted sensitive-data are introduced one by one. Taint sources are hardware input such as keyboard, network interface etc. The analysis is run under the assumption that analyzed applications should neither access nor process sensitive data. Once the application is launched, they monitor how sensitive data propagate within the system and build a flow graph based on their observation. If a node that corresponds to the analyzed application exists in the graph then it can be seen as malicious because it infringed the assumption made before.

In the presented work we use Blare [12, 13, 14, 15] an intrusion detection system parametrized by a security flow-policy. Blare uses a flow-policy that defines how information can legally spread within the system. Blare monitors information flow and checks if occuring flows are legal or not regarding the policy. Flows that violate the enforced policy are considered as signs of an intrusion and cause alerts. To monitor information-flows, Blare intercepts syscalls and thus deduces an over approximation of occuring information flow at system level. For example, when a process P reads a file F, it performs a syscall. Blare intercepts this syscall and deduces that information flows from F to P. On Linux and Android, Blare uses the LSM framework [16] that introduced hooks in kernel-code to intercept system calls. Besides intercepting syscalls, Blare also performs a finer monitoring in the binder driver on Android. The binder is a mechanism used for Inter-Process Communication (IPC) in Android and that has been slightly modified for Blare. The modification allows to get details about the sender and receiver of binder-related IPCs and thus to deduce the occurring flow during such IPC. To enforce a flow policy Blare uses two labels (or tags) attached to each container of the system (file, processes, socket). Their values are made of identifiers (integers), each of them corresponds to unique pieces of information. The identifiers are chosen during the design of the flow-policy where the policy-designer identifies the pieces of sensitive information whose flow should be monitored. The first tag is named *itag* and indicates the origin of the current content of an object. More precisely, when the value of the *itag* attached to a container c (which is denoted by $itag(c)$) equals $\{i_1...i_n\}$, it means that the current content of c has been computed from at least pieces of information identified by $i_1...i_n$. Blare updates the value of the *itag* attached to an object each time it considers that the content of this object has changed. The second tag is named *ptag* and defines sets of authorized contents of a container. A *ptag* is a collection of set of identifiers and its value is defined during the policy design step. The default value of *ptag* is null. It means no pieces of information that are assigned an identifier can flow to the container. These two tags permit to check if the content of a container c is legal or not regarding the flow policy. A container c having an $itag(c)$ and a $ptag(c)$ has a legal content (and thus the last information flow that has modified its content) is legal if and only if there is an element e of $ptag(c)$ such that $itag(c) \subseteq e$. In the following we present how to enhance the benefit brought by the information flow monitor. In particular we present how we can use its logs to reconstruct a system graph of a suspicious application in order to learn about its behaviour.

System flow graph for diagnosing

System flow graph definition. In this work we aim to construct what we call the *system flow graph* of an execution. The *system flow graph* of an execution describes how a running code disseminates its own pieces

of information in the operating system. A *system flow graph* is a directed graph $G = (V, E)$ where V and E are respectively the nodes and edges of G. Each node $v \in V$ corresponds to a container of information of the operating system, each edge $e \in E$ corresponds to a unique flow of information from a source container to a destination container. Nodes are labelled by the container type (file, process or socket), the name of the container, the system identifier of the container (inode or pid). Edges are labelled by the idenfiers of the pieces of information involve in the flow and by the timestamps that correspond to system date.

System graph construction The execution of applications induces a lot of information flows in the system depending how the different processes have accessed or communicated with other files, socket or processes of their execution environment. To learn about the information flow induced by a particular execution we propose to use only one information identifier i. This particular value is used to compute the value of the itag attached to the code that will be executed. On Android, each application comes as a single package, the *apk*. We choose to consider it as the source because it contains all the resources used by the application including its own code. The itag of the file *apk* is set to i and any other container has an itag equal to *null* at the beginning of the experiment. All the containers have a *ptag* value equal to *null* then any information flow involving a piece of data coming from the monitored application will lead to an alert. Once the application is launched, we use it as a normal user would do. While the application is running, Blare monitors information flows and logs all the alerts. An alert in Blare log has the format described in figure 1.

The alert is divided in four parts. The first part contains a timestamp and a label `BLARE_POLICY_VIOLATION`. The timestamp indicates when the alert was raised by Blare. The label helps to differentiate Blare messages from others in kernel messages. The remaining parts, separated by the character '>', describe a flow: the source, the destination and the identifiers of the origin of information that Blare considers as propagating. Information given about source and destination containers are their type, their name and their identifiers. In the example describes in figure 1, Blare raises an alert as process *cat* illegally reads data computed from 1 and 2 in file *secret.txt*.

We build a directed graph $G = (V, E)$ corresponding to the propagation of information computed from i. Each node $v \in V$ corresponds to a container involved in an alert that Blare raises in previous step. Each node is labelled with three attributes: the type of the container, its name and its identifier in the system. Each edge $e \in E$ corresponds to a unique flow that causes an alert in previous step. What makes a flow unique are its source, its destination and the identifiers attached to the flow. Edge attributes are the identifiers attached to the flow and the timestamps of the alerts corresponding to this flow. We build the graph by parsing one by one each record of the log. Each time we encounter a new container, we add a new node to G whose attributes are the the name, type and id of the container. Each time we encounter a new flow, we add the corresponding edge to G. The source and destination of the new edge are the nodes that respectively correspond to the source and destination of the flow. If the flow is not new, we append the timestamp of the alert to the timestmap-attribute of the edge. We developed a tool that automatically parses the log and build the corresponding graph. Figure 2 is an example of a dot-graph that our tool can build. Boxes represent files and ellipses represent processes. The node label is the name of the container. The edge label contains 3 pieces of information: pieces of information that are propagating, occurrence of the flow and the first time the flow was observed.

System flow graph analysis for diagnosis A system flow graph provides to an expert the external behaviour of an execution of a suspicious application and gives an early diagnosis in case of attacks. The expert can learn which containers of information are impacted by an execution in listing the node of the system flow graph. Thus he can verify that no sensitive containers are touched by the execution. In case of a malware, the expert thus learns where the malware has dumped its payload. If the expert finds a container impacted by the execution that should not be, he can use the system graph to deduce how it happened. To do that, we collect paths from the node of the application package to compromised nodes. These paths precisely describe how information walk from the suspicious application to the target.

Case study: DroidKungFu

Researchers first detected this malware [17] embedded in legitimate applications on Chinese alternatives of *Google Play*. Based on the report they made, we know the application embeds malicious code: root exploits and an additional application that is meant to be installed in system partition. We analyse here a sample of this malware. Its developer published it as a SIP-client to lure users who are looking for such application. According to our methodology, we tag the application package, monitor flows of information coming from it and build the corresponding graph to analyse it. The identifier we use for the information coming from the application is 99. Figure 2 is an extract of the graph we got. The entire graph contains 245 edges (245 unique flows) and 95 nodes. Red edges are the most suspicious as information obtained from 99 spread to different containers which three of them belong to the system (containers in */system* and */proc*). For these system containers, the paths from 99's source to them indicate that a native application comes with the application we analyse and dumps data in these containers. These pieces of data are applications, a native binary and an *apk* in */system*, and raw data in */proc/sys/kernel/hotplug*.

[TIMESTMAP][BLARE_POLICY_VIOLATION] SRC_TYPE SRC_NAME SRC_ID > DEST_TYPE DEST_NAME DEST_ID > $\{i_1 ... i_n\}$
General format

[10000][BLARE_POLICY_VIOLATION] FILE SECRET.TXT 18 > PROCESS CAT:CAT 2007 > $\{1, 2\}$
Example

Figure 1: Blare log records

According to report made on DroidKungFu [17], this flow corresponds to a privilege escalation that tries to exploit a vulnerability in the kernel. For the two other red flows, */system/app/com.google.ssearch.apk* to *system_server* and *system_server* to */data/system/packages.xml*, what make them suspicious is that they happen after the dump of *com.google.ssearch.apk* to */system*. If we combine our knowledge about Android inner-working with the diagnosis given by the path made from these two red edges, we can deduce that the *apk* file is installed as a system application on the device. By doing the same with the paths made of blue edges, we can deduce that the newly installed application is automatically launched. To sum it up, we found that the SIP-client we installed dumps two applications, a binary and an *apk*, in system partition. Furthermore, by analysing paths from the *apk* we also found that the application was automatically installed as a system application and run. The advantage of the system application is that the user cannot uninstall it.

Conclusion

This article proposes a new structure called *system flow graph* that describes how a particular execution is responsible of information dissemination in the system. We also explain how such structures can be constructed using the information flow monitor Blare. Lastly we describe how *system flow graphs* are helpful to give an early diagnosis in case of attacks. We evaluate our approach in an experiment during which we study a malicious application published on alternatives of *Google Play*.

In future work we plan to use *system flow graphs* in intrusion detection since a set of *system flow graph* of different executions of a suspicious application will naturally lead to a model of application behaviour.

References

[1] J. Network, "2011 mobile threats report," http://www.juniper.net/us/en/local/pdf/additional-resources/jnpr-2011-mobile-threats-report.pdf?utm_source=promo&utm_medium=right_promo&utm_campaign=mobile_threat_report_0212.

[2] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, , and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *In Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[3] T. Vidas, D. Votipka, and N. Christin, "All your droid are belong to us: a survey of current android attacks," in *Proceedings of the 5th USENIX conference on Offensive technologies*. Berkeley, CA, USA: USENIX Association, 2011, pp. 10–10.

[4] http://googlemobile.blogspot.fr/2012/02/android-and-security.html.

[5] N. Percoco and S. Schulte, "Adventures in bouncerland, failures of automated malware detection within mobile application markets," TrustWave SpiderLabs, Tech. Rep., 2012.

[6] J. Oberheide and C. Miller, "Dissecting the android bouncer."

[7] R. Paleari, L. Martignoni, G. F. Roglia, and D. Bruschi, "A fistful of red-pills: how to automatically generate procedures to detect cpu emulators," in *Proceedings of the 3rd USENIX conference on Offensive technologies*. USENIX Association, 2009.

[8] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making information flow explicit in histar," in *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*. Berkeley, CA, USA: USENIX Association, 2006, pp. 263–278.

[9] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris, "Information flow control for standard os abstractions," in *Proceedings of the 21st Symposium on Operating Systems Principles*, Stevenson, WA, October 2007.

[10] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "PiOS: Detecting Privacy Leaks in iOS Applications," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2011.

[11] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: capturing system-wide information flow for malware detection and analysis," in *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*. New York, NY, USA: ACM, 2007, pp. 116–127.

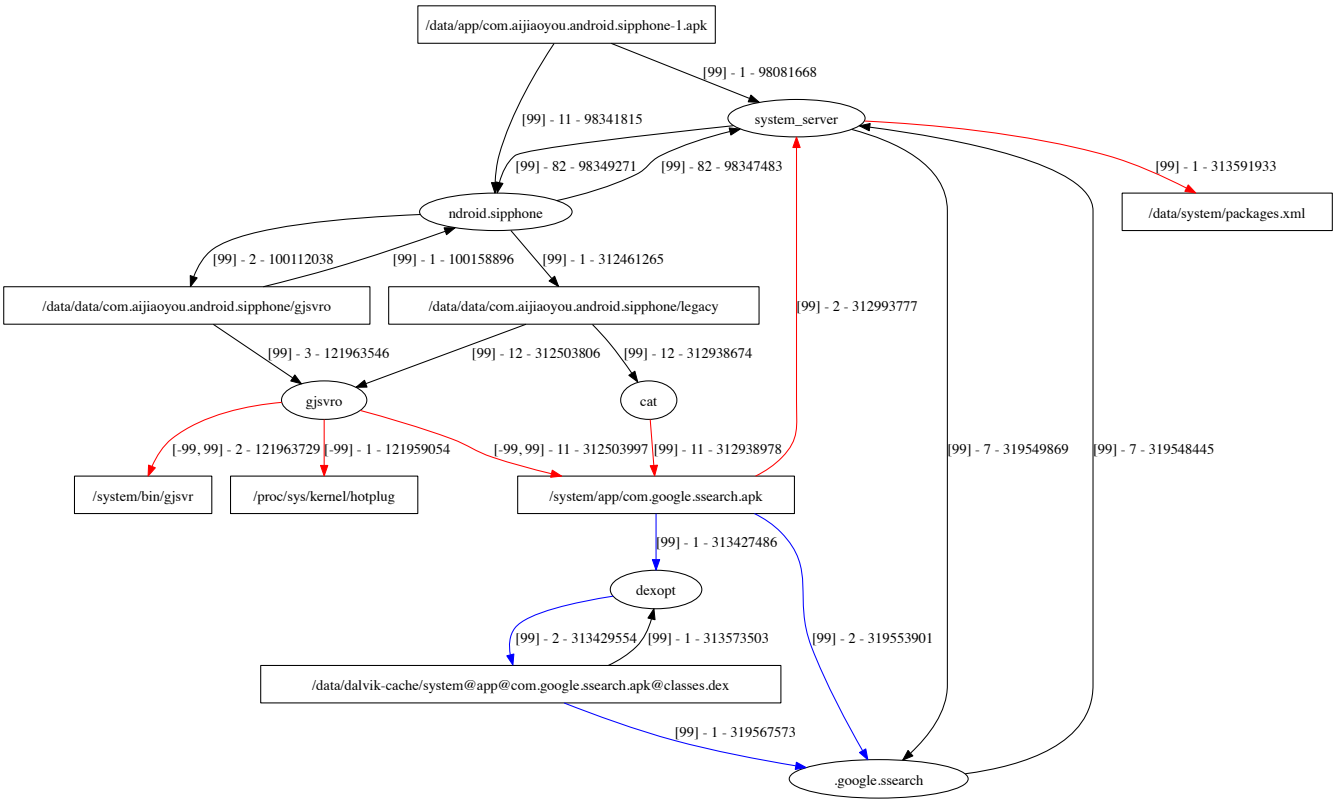

Figure 2: An example of flow-graph built from observed information-flow in the system

[12] G. Hiet, L. Mé, B. Morin, and V. Viet Triem Tong, "Monitoring both os and program level information flows to detect intrusions against network servers," in *IEEE Workshop on "Monitoring, Attack Detection and Mitigation"*, 2007.

[13] S. Geller, C. Hauser, F. Tronel, and V. Viet Triem Tong, "Information flow control for intrusion detection derived from mac policy," in *ICC 2011*, 2011.

[14] C. Hauser, F. Tronel, J. Reid, and C. Fidge, "A taint marking approach to confidentiality violation detection," in *10th Australasian Information Security Conference (AISC 2012)*. Australian Computer Society, 2012.

[15] R. Andriatsimandefitra, S. Geller, and V. Viet Triem Tong, "Designing information flow policies for android's operating system," in *Proceedings of the IEEE Internation Conference on Computer Communications (ICC)*, 2012.

[16] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman, "Linux security module framework," in *OLS2002 Proceedings*, 2002.

[17] K. McNamee, "Malware analysis report trojan:androidos/droidkungfu.a."